On Distributing the Runtime of the Chemical Programming Model

Marko Obrovac, Cédric Tedeschi

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

On Distributing the Runtime of the Chemical Programming Model

Marko Obrovac — Cédric Tedeschi

N° 7661

June 2011

ISSN 0249-6399 ISRN INRIA/RR--7661--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

On Distributing the Runtime of the Chemical Programming Model

Marko Obrovac , Cédric Tedeschi

Theme :
Équipe-Projet Myriads

Rapport de recherche n° 7661 — June 2011 — 16 pages

Abstract: Internet is nowadays a global computing platform comprising myriads of autonomous services which require targeted composition and coordination. Nature-inspired, and more specifically *chemical* programming models, in which a computation is seen as a set of reactions between molecules interacting freely in a solution, has emerged as a promising alternative for programming such platforms. While much works recently highlighted the versatility and expressiveness of such a model, its distributed execution is still a widely open problem. With this paper, we start the study of a distributed execution environment for chemical programs. We propose a framework based on a peer-to-peer network on top of which molecules and reactions are distributed. We exhibit some optimality properties of our algorithms. A real-world prototype has been developed, and deployed over the nation-wide Grid'5000 testbed. These experiments confirm the viability of our proposal.

Key-words: Chemical computing, Distributed Hash Table, Execution tree

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Vers l'exécution distribuée de programmes chimiques

Résumé : Internet est devenu une plate-forme de calcul globale dans laquelle une myriade de services autonomes sont composés dynamiquement en fonction des besoins de millions d'utilisateurs simultanés. Face à ce nouveau changement des plates-formes de calcul, il est nécessaire de proposer de nouveaux modèles de programmation. Les modèles de programmation inspirés par la nature, et en particulier le modèle chimique montre des propriétés intéressantes pour modéliser de telles interactions. Dans ce modèle, les services (encapsulant par exemples des composants logiciels, des données, ou des capteurs) sont vus comme des molécules dont les interactions sont modélisés par des règles de réactions.

Alors que beaucoup de travaux récents autour de ce paradigme ont mis en évidence l'intérêt de ce modèle, son exécution distribuée reste un problème largement ouvert. Dans ce rapport, nous proposons et implémentons un modéle d'exécution pour ce modèle. Nous nous appuyons sur une table de hachage distribuée à travers laquelle les molécules sont distribués, et au-dessus duquel un arbre d'exécution est construit.

Nous avons développé un prototype de notre approche et l'avons expérimenté sur la plate-forme Grid'5000, fournissant une preuve de concept et de sa viabilité à large échelle.

Mots-clés : Calcul chimique, Table de hachage distribuée, Arbre d'exécution

Contents

1 Introduction

Internet as we have used it over the last decade seems to come close to an end. Web *surfing* represents today a minor part of Internet's traffic. The *Internet of Services* has now emerged as a global computing platform, under the shape of a digital ecosystem where myriads of *services* are composed dynamically to satisfy billions of requests constantly sent by users at the edge of this global platform, through applications. A service can encapsulate various components such as sensor devices, storage space, computing power, or software, making them available through the network. They are dynamically combined into composite services (workflows) usually presenting a high degree of parallelism and distribution. To leverage such high level abstractions, new programming models are required, able to express autonomic coordination, parallelism and distribution.

Nature-inspired models recently regained momentum in this context [14]. In particular, the *chemical* programming paradigm, initially developed to naturally express parallel processing [4], offers intuitive and simplified ways to express a wide range of coordination problems [7]. Its expressiveness allows programmers to directly tackle an application's logic and behaviour, while abstracting out the details of inherent sequentiality of the target specific platform or language. Within this model, a computation is seen as a chemical solution where molecules of data react according to some rules (programs) to produce new data. Reactions take place in an implicitly autonomous, parallel and distributed fashion until the state of inertia, where no more reactions are possible. The solution then contains the result of the computation. Following this model, the Higher-Order Chemical Language (HOCL) goes further by providing the higher order [5]: any digital entity can be represented as a molecule interacting with others. Services, data, and even users can interact this way. More importantly, rules can react with other rules, programs dynamically modifying programs.

While the expressiveness and versatility of the chemical paradigm have been established, the distributed execution of chemical programs is still a widely open problem, hindering the model to be actually leveraged.

The execution of a chemical program is comprised of three main phases: finding reactants, performing the reactions, and detecting the inertia. The few isolated attempts at tackling the problem over a distributed platform had limited impact, for they focused on some particular topologies or cases. Our goal is to address this problem with three main requirements in mind: (i) **Parallelisation and Distribution Transparency.** The runtime of a chemical program is implicitly parallel and distributed. Parallelisation and distribution should remain invisible to the programmer. (ii) **Platform Independence.** In the same vein, our objective is to build a distributed environment to be executed on top of any distributed platform, by abstracting out potential hardware or communication heterogeneities. (iii) **Scalability.** Finding reactants at large scale, and, similarly, detecting that no more reactants in the platform can react is a typically hard task, that, at large scale should be tackled through intensive parallelisation and distribution.

Contribution. The emergence of peer-to-peer technologies for information retrieval leads to unexplored ways to solve the problem. In this paper, we propose a scheme building an execution tree on top of a distributed hash table (DHT) [1, 12] as a framework for the problem. We exhibit a distributed algorithm for inertia detection which we show optimal in terms of number of tests. We developed a distributed chemical machine prototype on top of the FreePastry [2] DHT implementation. Experimentations conducted on the Grid'5000 platform [3] establish its viability, and thus open doors for the actual adoption of such a model.

The next section presents the chemical model and the challenges of a distributed runtime. Section 3 details and analyses our architecture and algorithms. Section 4 discusses the prototype and its experimental results. Section 5 explores related works. Finally, Section 6 concludes and advocates future work.

2 Chemical Programming Model

The chemical model was initially proposed for a natural expression of parallel processing, by removing artificial structuring and serialisation of programs, focusing only on the problem logic. Following the chemical analogy, data are molecules floating in a solution. They are consumed according to some reaction rules, *i.e.*, the program, producing new molecules, *i.e.*, resulting data. These reactions take place in an implicitly parallel and autonomous way, until no more reactions are possible, a state referred to as *inertia*. This model was first formalised by GAMMA [4], in which the solution is a multiset of molecules, and reactions are rewriting rules on this multiset. A rule **replace** P **by** M **if** V consumes a set of molecules N satisfying the pattern P and the condition V, and produces a set of molecules M. Let us consider the following example of a chemical program extracting the maximal value from a set of integers:

$$\textbf{replace } x :: int, y :: int \textbf{ by } x \textbf{ if } x \geq y \textbf{ in } \quad \langle 2, 4, 5, 7, 9 \rangle$$

The rule specifies that any pair of integers inside the solution react, consuming them to create a new integer with the highest value of the two. Note that here, the condition holds for any pair of integers. While the result of the computation is deterministic, the order of its execution is not; GAMMA simply ensures the mutual exclusion of reactions by the atomic capture of the reactants. In our example, a possible execution is the following (2 and 7, as well as 4 and 5, react first, producing the intermediate state):

$$\langle 2, 4, 5, 7, 9\rangle \rightarrow^* \langle 5, 7, 9\rangle \rightarrow^* \langle 9\rangle$$

Recently, the *Higher-Order Chemical Language* (HOCL) [5] raised chemistry to the higher order. In HOCL, any entity taking part in the computation, (service or platform) can be represented as reactants. Reaction rules themselves react with other rules. In other words, programs can modify programs at runtime. These aspects confer on HOCL an uncommonly high expressiveness, able to deal with a wide variety of coordination on large scale platforms, as shown in [7, 6, 8]. However, such a *chemical* coordination remained mostly conceptual until now.

Distributed Scenario. We here aim at providing a platform to execute any chemical specification. As an illustration, let us reconsider our *max* example. Our platform should be able, without any extra line of code, to execute it over a distributed platform, *i.e.*, distribute the three phases mentioned in Section 1. A typically possible scenario is the following. First, the solution (multiset) is distributed over the nodes, each node receiving a part of the integers, constituting their *local* solutions. This triggers the second phase, during which each node searches for reactants within its own local solution. Reactions take place separately on each node. When a local solution becomes inert, it needs to be merged with others to continue further the computation. Imagine two inert solutions, each containing an integer. These two integers need to meet to react in their turn, thus producing a new inert solution that will merge with others, and so forth until one node contains the final inert solution, *i.e.*, the maximal value of the initial multiset.

3 Distributed Chemical Platform

Our platform, illustrated by Figure 1, builds upon a distributed hash table (DHT) [1, 12], thus securing independence from the underlying environment, and partially solving the scalability issue as nodes are able to communicate efficiently no matter of their number. The external application sending its program to the DHT can contact any of the DHT nodes, thus facilitating naturally load balancing. The platform completes the requirement by employing the locality philosophy: computation happens first where the data is located. In the following, Pastry [12] is used to implement this overlay layer. Note that any DHT could fill this role.

We first detail how molecules are distributed and the computation initiated. Then, we focus on the distributed mechanisms required to find reactants and detect the inertia. We show, by providing a first naïve algorithm that suboptimality in terms of number of reaction tests is easily encountered if the

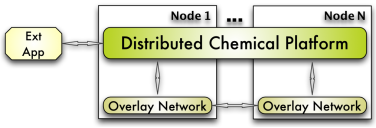

Figure 1: The platform.

algorithm is not designed properly. The optimal algorithm we designed is then given. Its optimality is formally proven.

3.1 Data and Execution Distribution

To trigger its runtime, the holder of a chemical program contacts one node it knows in the Pastry overlay (referred to as the *ring* in the following) and transfers it the solution. That contact node is referred to as the *source*, since it represents the data's entry point in the ring. As we will see, the execution will also finish on the source node, which will finally deliver the inert solution, *i.e.*, the result, to the requesting node. The source is the root of the execution tree whose construction we'll be now discussing.

Once the source node receives the data, it scatters the data molecules across the ring according to their hash values. (The cryptographic hash function of the underlying DHT guarantees uniform dispersion with high probability (w.h.p.).) Molecules are routed concurrently according to Pastry's routing scheme, in $O(\log n)$ hops [12]. In the course of the routing process, the path of each molecule is traced by intermediary nodes, called *forwarders*, from the source node to the molecule's destination node, referred to as the *worker*. By passing on molecules, forwarders maintain a *local state* (in addition to the Pastry's routing table) containing the set of nodes to which they forwarded a molecule. Note that forwarders, together with the source node, will be workers as well, w.h.p. Finally, the source node spreads one final message, *mc*, to nodes in its own local state, containing the rules to execute.

Upon the receipt of *mc* on a node p from a node s, p completes its state with s, referring to it as its *parent* (in the multicast tree being built). In case p has already received *mc* from another node, it just drops it, but sends another specific message back to s, which, upon receipt, deletes p from s' local state. This ensures that, combined, the local states form a tree. This tree, rooted at the source node, will be used later to make partial inert solutions move backwards to the source node. The tree thus created presents some similarities with the Scribe publish/subscribe system [9]. Note that the complexity of the local state is logarithmic to the number of nodes in the system, as nodes referenced in the local state of a node (except its parent) are necessary inside Pastry's routing table, itself logarithmic in size.

After receiving the last multicast, nodes start locally the computation. Every possible combination of molecules residing on a node is checked on this node against the rules, and, if possible, reactions take place. The combinations' cardinality is determined by the number of molecules local to the node and the number of a rule's arguments. When the part of the solution received by one node is inert, it must assiociate itself with other nodes to continue the computation.

Each of them sends its inert local solution to the respective parent. The parents then merge them with their own and continue the computation. Only when the parent has received all of its children's solutions and when its local solution is inert, the process continues with the parent transferring its local solution to its parent, and so forth until all of the inert local solutions reach the source node, which delivers the global solution after executing it until inertia.

Fault Tolerance. While failures can affect our scheme (failures or disconnections of nodes can lead to (i) routing problems, and (ii) loss of molecules), it is not our primary concern here. However, we give a few hints for its reliability. The sub-tree formerly rooted at a crashed node becomes unable to forward its results up in the tree. Inspired by the work in [9], a simple detection and reconnection protocol can be defined: when initiating the last broadcast message, the source can include its ID. Then, when a node is unable to reach its parent, it can dynamically find a new path to the root by launching a reconnection request in the DHT on the source ID, and thus rebuild a connected tree. Dealing with the loss of molecules, one can rely on replication [13].

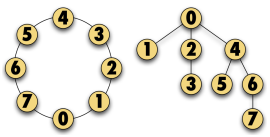

Figure 2: The original ring and the dissemination tree.

Execution Example. Consider an eight-node ring, as shown on the left part of Figure 2. For the sake of simplicity, let node 0 be the source node and let the program contain eight molecules whose hash identifiers match exactly those of the nodes, and node 0's routing table contains nodes 1, 2, and 4. Node 0 will send molecules 1, 2 and 4 directly to their respective workers, adding their identifiers in its local state. It transmits the molecule 3 via node 2, which adds node 3 it in its local state, and so on until all molecules have reached their worker. Node 0 multicasts the rules to its local state (nodes 1, 2 and 4). Node 1 starts the execution, while nodes 2 and 4 forward the rules to their children before, completing the tree on the right part of Figure 2. When their local solutions become inert, the leaf nodes (1, 3, 5 and 7) send them to their parents. After merging the newly arrived local solution and executing the set, the inner nodes transfer their local solutions to their parent, and so on until the solution reaches the source node.

3.2 Reaction Condition Checking

We now focus on reaction condition checking and inertia detection. We developed two algorithms distributing the task of trying every possible combination of molecules. The first one is an intuitive but sub-optimal method, used to highlight the fact that, if one is careless, many unnecessary tests can be done, increasing again the complexity of an already hard task. The second one, referred to as *BucketSolver* is shown to be optimal in terms of number of combination tests.

A Naïve Algorithm. In a first intuitive approach, upon the receipt of molecules from one of its children, a node starts a *computation cycle*, during which every possible molecule combination of the local solution is tested, possible reactions performed, and thus local inertia reached. Provided a parent has got g children, there are exactly $g + 1$ such cycles: the first one occurs after the initial dissemination of molecules, while the other g cycles take place after the result of each child has been received. Note that, even though two or more children's results might be received simultaneously, this will not reduce the number of computation cycles in terms of number of tests. As will later reveal the analysis, the total number of tests performed by this algorithm depends on the number of nodes in the tree as well as its structure, and is considerably higher than the optimal number of combination tests.

3.2.1 BucketSolver Algorithm

The sub-optimality of the naïve algorithm comes from the conception of the reaction condition checking routine. Once a node receives or generates new molecules, it merges them in its unique local solution without keeping track of already checked combinations, leading to future unnecessary tests.

When a node transfers its local solution to its parent, the parent is sure that all of the combinations in the node's local solution have already been tried. The parent does not really need to know exactly which combinations have been checked, as long as it knows the set of molecules they derive from. Thus, we create *buckets* into which we put sets of molecules whose combinations have already been tried.

When a node originally receives molecules from the source, it puts them each in its own bucket. A computation cycle comprises checking only inter-bucket combinations — those whose elements belong to different buckets. For the sake of clarity, let us consider two buckets. Formally, when checking a combination of r arguments, j, $0 < j < r$, elements are picked from bucket a, while $r - j$ elements are picked from bucket b. If the combination is evaluated positively, the elements are removed from their respective buckets and once the reaction has been carried out, each resulting molecule is placed in a new, separate bucket.

Once two initial buckets' intersection combinations have been checked, they are *fused* — molecules from one bucket are put into the other and the now empty bucket is deleted. As shown later, the act of fusion guarantees all of the intra- and inter-bucket combinations will be examined. Following this logic, the solution, be it local or global, is declared to be inert once there is only one bucket left in the system.

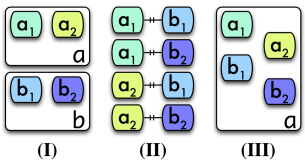

Figure 3: The checking buckets process.

Consider the example illustrated in Figure 3. Imagine a node checked two molecules, a_1 and a_2, which now reside in bucket a. The node then receives a

result from its child and creates a bucket b (Figure 3(I)). Now, it checks all of the combinations except those whose elements all reside in the same bucket — all the combinations are tested but (a_1, a_2) and (b_1, b_2) (Figure 3(II)). Finally, presuming no reaction took place, the two buckets are merged into one containing all of the elements (Figure 3(III)). This algorithm is valid for $r \geq 2$, since at least one element per bucket must be picked.

In presence of one-argument rules, condition checks only need to be performed on the molecules received at the time of the initial dissemination. The ones transmitted from children are only forwarded to parents, as they were already checked by their initial worker, located in the sub-tree.

BucketSolver provides *inertia detection* (all of the combinations will be examined) while being *optimal* (every combination will be checked only once). These assertions are proved in Section 3.2.2 and corroborated with experimental results in Section 4.

3.2.2 Analysis

We now present an analysis of the number of combinations examined to detect the inertia of both algorithms presented and establish the optimality of BucketSolver. Obviously, this analysis assumes the solution to be already inert, as we here *detect* inertia. Note that a complexity analysis of an unstable solution is not feasible since the number of reactions and nodes can vary according to the rules and molecules being checked.

The number of nodes is denoted by n. The program considered contains m molecules. Thus, after the initial dissemination detailed in Section 3.1, every node holds approximately m/n molecules. We assume, for the sake of simplicity, and without loss of generality, that the program considered now is composed of one commutative rule with r arguments. In other words, unordered combinations of r molecules are checked against the condition of this rule. Under these circumstances, the minimum number of combination to be checked is $N_{min} = \binom{m}{r} = \frac{m!}{r!(m-r)!}$. To quantitatively compare the two approaches, we assume the execution tree is a full g-ary tree, where $g \geq 2$. This allows us to analyse and differentiate the approaches clearly by taking advantage of the symmetry and recursion of full trees. Nevertheless, we prove that the proposed algorithm keeps its efficiency when applied to any type of tree, be it full or not.

Let us firstly analyse the naïve algorithm. On a given depth i, $0 \leq i \leq d$, the total number of combinations checked by one node is:

$$N_{i_1} = \binom{\frac{m}{n}}{r} + \sum_{k=1}^{g} \binom{km_{ci} + \frac{m}{n}}{r} = \sum_{k=0}^{g} \binom{km_{ci} + \frac{m}{n}}{r} \quad (1)$$

where m_{ci} is the number of molecules received from the child at depth $i+1$ and is defined as:

$$m_{ci} = \frac{m}{n} * \frac{1}{g-1} \left(g^{d-i} - 1 \right)$$

Summing by depth all of the checks made, we get the total number of checks $N = \sum_{i=0}^{d} g^i * N_{i_1}$, which yields, after introducing Equation 1 and arranging the terms:

$$\begin{aligned} N &= \binom{m}{r} + n\binom{\frac{m}{n}}{r} \\ &+ \sum_{i=0}^{d-1}\left[g^{i+1}\binom{m_{ci}}{r} + g^{i}\sum_{j=1}^{g-1}\binom{jm_{ci}+\frac{m}{n}}{r}\right] \end{aligned} \tag{2}$$

This establishes the sub-optimality of the naïve approach. Following is the analysis which shows the optimality of BucketSolver.

Lemma 1. *When fusing two buckets, a and b, the number of combinations totals to $N_{ab} = \binom{m_a+m_b}{r}$ where m_a and m_b represent the number of molecules contained in each of the two buckets.*

Proof. Given that each of the buckets' combinations have already been checked, we have to prove their intersection is checked also, *i.e.*

$$N_{a\cap b} = \binom{m_a+m_b}{r} - \binom{m_a}{r} - \binom{m_b}{r} \tag{3}$$

The combinations expressed in Equation 3 are found by picking r elements from the set $a\cup b$ where j elements come from bucket a and $r-j$ elements from bucket b $(0 < j < r)$. When summing the number of combinations, we get the following:

$$\sum_{j=1}^{r-1}\left[\binom{m_a}{j} * \binom{m_b}{r-j}\right] = \binom{m_a+m_b}{r} - \binom{m_a}{r} - \binom{m_b}{r}$$

□

Following Lemma 1 we observe that the necessary and sufficient condition that all of the combinations have been checked is having exactly one bucket left, since when there is only one bucket present, there are no inter-bucket combinations to be examined, which, by definition, means all of the possible combinations in the bucket have been checked.

Lemma 2. *The number of combinations checked by BucketSolver in a full g-ary tree amounts to $N = \binom{m}{r}$.*

Proof. In the first computational cycle, every combination will be checked: $N_{i_1} = \binom{\frac{m}{n}}{r}$. Each time a node receives a result from one of its children, it checks $N_{a\cap b}$ combinations, where bucket a represents the molecules already present on the node and bucket b represents the incoming result of a child. A node does so g times, and the number of combinations on it totals to:

$$N_{i_1} = \binom{gm_{c_i}+\frac{m}{n}}{r} - g\binom{m_{c_i}}{r}$$

which, when summed by depth yields $N = \sum_{i=0}^{d} g^i N_{i_1} = \binom{m}{r}$.

□

Lemma 3. *The number of combinations examined by the modified one-rule-argument BucketSolver algorithm is $N = \binom{m}{1} = m$.*

Proof. Only one computation cycle takes place upon the initial molecules dissemination. The number of combinations processed by each node is $N_{i1} = \binom{\frac{m}{n}}{1} = \frac{m}{n}$. Provided there are no more computation cycles, the number of combinations tested amounts to m. □

Following Lemmas 2 and 3 we have:

Corollary 1. *BucketSolver checks every combination only once.*

Theorem 1. *Lemma 2 holds true also for non-regular trees.*

Proof. The number of combinations checked by a node i is:

$$\begin{aligned} N_i = & \binom{\frac{m}{n}}{r} \\ + & \sum_{k=0}^{g_i-1} \left[\binom{\frac{m}{n} + \sum_{j=1}^{k+1} m_j}{r} - \binom{\frac{m}{n} + \sum_{j=1}^{k} m_j}{r} - \binom{m_{k+1}}{r} \right] \\ = & \binom{m_i}{r} - \sum_{j=1}^{g_i} \binom{m_j}{r} \end{aligned}$$

where g_i is the number of node i's children, m_j is the number of molecules forwarded to it by its child j and m_i is the final number of molecules which node i will send to its parent: $m_i = \frac{m}{n} + \sum_{j=1}^{g_i} m_j$. Naturally, m_i will become one of node i's parent's m_j, which means they will cancel out one another. Consequently, the total number of combinations checked by the algorithm is: $N = \sum_{i=1}^{n} N_i = \binom{m}{r}$ □

This analysis shows that both algorithms insure the detection of inertia. It also shows that BucketSolver checks every combination only once, and is, thus, optimal. Section 4 experimentally confirms this statement.

4 Evaluation

We developed a prototype of our architecture and algorithms[1], described below. It was tested on a simple chemical program outlined in section 4.1. Experimental results are finally presented.

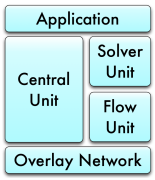

Figure 4: The prototype's logical concept.

The logical concept of the prototype is depicted in Figure 4. The central and flow unit represent the implementation of the architecture laid out in Section 3.1. The central unit is in charge of communicating with external applications and sending and receiving molecules, while the flow unit builds a node's local state.

The central role in the execution is played by the solver unit, which is the implementation of the inertia-detection algorithms, and has, thus, two different implementations. The implementation of the naïve algorithm reflects precisely its theoretical description, while that of BucketSolver carries a slight refinement — a multi-threaded version of the algorithm is implemented.

[1]Sources are available in the *branches/devel-distrib* directory of the *svn* repository located at *http://gforge.inria.fr/scm/?group_id=2125*.

4.1 Test Program

The evaluation of the proposed architecture and algorithms was conducted using a simple chemical program containing one multiset of 5000 molecules composed of two integer numbers - an index and a value associated with it, and a single rule, *sort*, operating on them:

$$\begin{aligned}\textbf{let } sort = \ & \textbf{replace } \{x.i, x.v\}, \{y.i, y.v\} \textbf{ by } \{x.i, y.v\}, \{y.i, x.v\} \\ & \textbf{if } (x.i > y.i \ \&\& \ x.v < y.v) \,||\, (x.i < y.i \ \&\& \ x.v > y.v)\end{aligned}$$

The rule consumes two molecules if they are not already sorted in ascending order. Two new molecules are then created, holding the same indices as the original ones, but with swapped values. Although remarkably simple, this program exhibits a few pertinent properties. Firstly, it keeps the number of molecules in the solution, as well as the complexity of the program, constant, which means that, at the end of the computation, the multiset to be processed by the source node is important, constituting the potential scalability limit of our approach. Then, if our system can deal with such a program, it will *a fortiori* be able to run programs whose multiset's size decrases over time.

4.2 Experimental Results

Preliminary experiments were carried out on the prototype with the simple program presented above. The experiments were conducted on the nation wide Grid'5000 [3] platform, on machines equipped with two quad-core Intel Xeon E5520 processors, 24 GB of RAM and an InfiniBand 40G Ethernet card. In each experiment, the number of participating nodes varies from 1 to 500. Each processor core executes one logical node.

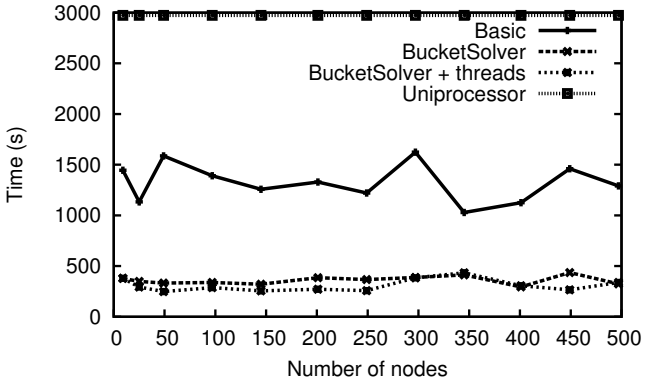

Figure 5: Execution time of test program.

Experiment 1 (Execution Time). Firstly, we examine the viability of our approach. Figure 5 shows the execution time of the test program. The flat *uniprocessor* line represents the execution time of the original HOCL compiler, artificially pulled all along the x-axis for the sake of comparison. The tests

confirm the architecture's viability, and considerable speedups are obtained: $S_{basic} \in [2, 3]$ for the naïve algorithm and $S_{bucket} \in [6, 12]$ for BucketSolver. The tests expose a considerable impact of the structure of the tree on the basic algorithm's performance, visible in the high fluctuation in execution time. BucketSolver is more resilient to structural changes and exhibits better performance results. Finally, using threads decreases further the total execution time for an average of 20%.

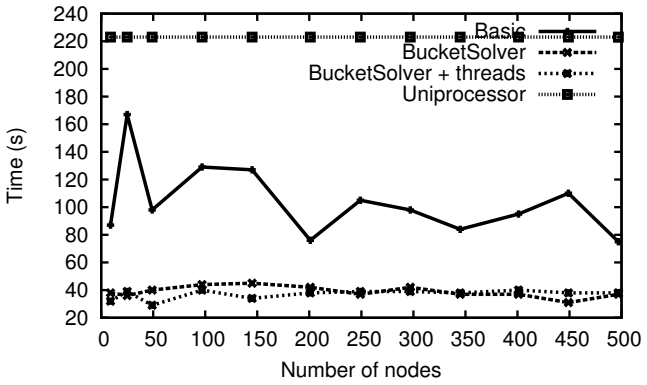

Figure 6: Execution time on an inert solution.

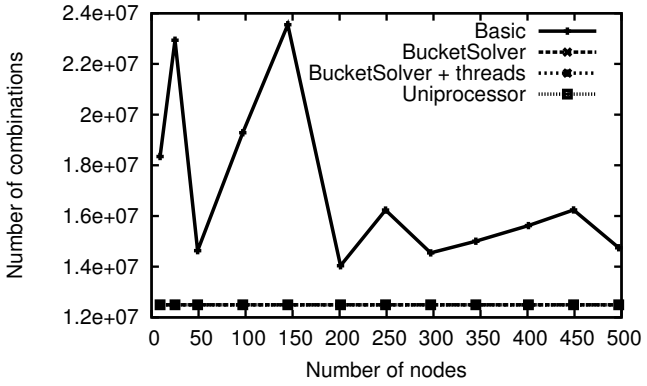

Figure 7: Number of checks done on an inert solution.

Experiment 2 (Inertia Detection). Next, we investigate the overhead of distributed inertia detection. The tests were conducted on an inert solution. The total execution time, depicted in Figure 6 suggests that the speedup for the basic algorithm stayed roughly the same ($S_{basic} \in [1.3, 3]$), whereas the speedup of BucketSolver decreased ($S_{bucket} \in [5, 7.8]$). This decline happens because there is no reaction distribution, only the condition checking is distributed. The total number of combinations checked, shown on Figure 7, reveals the rationale behind the difference in speedup of the two compared algorithms, and confirms BucketSolver's optimality. Due to the factorial complexity of the naïve

algorithm's sub-optimality, as much as twice the minimal number of checks is done.

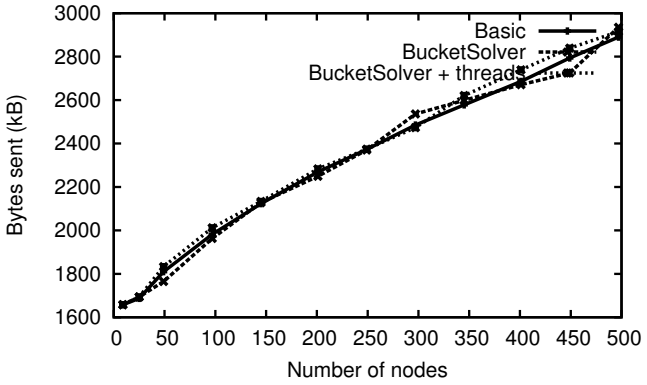

Figure 8: Cumulative communication cost.

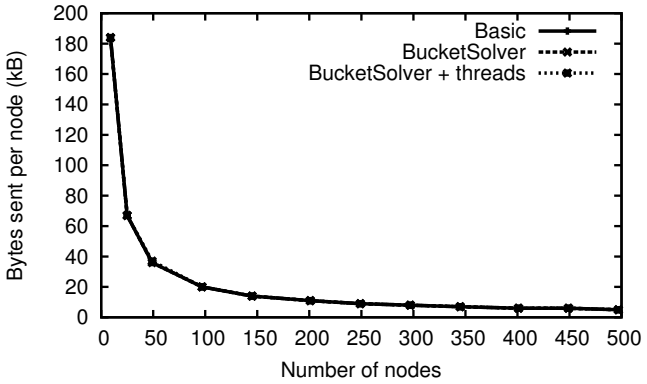

Figure 9: Communication cost per node.

Experiment 3 (Communication Overhead). Finally, we analyse the scalability related to communication, depicted in Figures 8 and 9. The former represents the total number of bytes sent during execution, while the latter illustrates the average communication cost per node. Figure 8 shows that BucketSolver does not induce an increase in communication costs. Furthermore, the total amount of data sent grows linearly with the number of nodes at the rate of approximately 2.5 kB per node added. Figure 9 shows that the average local communication costs of each node decrease as nodes are added, asymptotically approaching 2.5 kB, confirming the system's scalability in terms of communications.

5 Related Works

Surprisingly, in spite of the fact that the model is implicitly parallel, not much work has been done on parallel execution of such programs. The pioneering work of Banâtre *et al.* [4] provides two implementation methods, whose basic idea is a parallel machine. Each processor holds a molecule and compares it with the molecules of all the other processors. Two algorithms are proposed: (a) a synchronous one, where a centralised controller triggers each comparison step, and (b) an asynchronous one, in which the molecules travel along a vector of processors, either until they react, or until they have returned to their starting point. This last algorithm was implemented on top of an iPSC hypercube with 16 processors. In the work by Linpeng *et al.* [10], a program is executed on MasPar MP1, a massively parallel machine, using the fold-over operation. The molecules are placed on a strip and folded over after each vertical comparison. At each step, the elements in the upper segment of the strip are compared in parallel to those in the lower segment. Recently, Lin *et al.* developed a parser of GAMMA programs for their execution on a cluster exploiting GPU computing power [11]. Although these works present significant speedups, the targeted platforms are quite limited. On our side, we envisage a hardware independent open platform.

6 Conclusion

Chemistry-inspired models were recently highlighted as an alternative model for the Internet of Services. However, its distributed runtime remains an open issue.

This paper proposes a distributed execution platform for chemical programs. One contribution here is its generality in the sense that (a) any chemical program is automatically distributed and executed on it, and (b) it is platform-independent. Detecting inertia in a distributed context poses certain algorithmic challenges. Our contribution here is an optimal distributed approach to it. The proposal is experimentally validated on a real-world test-bed, establishing its viability.

Future work will be focused on facing high workloads. A tree reorganisation scheme is being formulated, that will allow the system to control the tree structure and, thus, precisely balance the workload.

References

[1] Chord. http://pdos.csail.mit.edu/chord/ (Jun 2011)

[2] Freepastry. http://www.freepastry.org (Jun 2011)

[3] Grid'5000. http://www.grid5000.fr (Jun 2011)

[4] Banâtre, J.P., Coutant, A., Le Metayer, D.: A Parallel Machine for Multiset Transformation and its Programming Style. Future Gener. Comput. Syst. 4 (1988)

[5] Banâtre, J.P., Fradet, P., Radenac, Y.: Generalised Multisets for Chemical Programming. Mathematical Structures in Computer Science 16 (2006)

[6] Banâtre, J.P., Fradet, P., Radenac, Y.: Towards Chemical Coordination for Grids. In: SAC. pp. 445–446 (2006)

[7] Banâtre, J.P., Priol, T.: Chemical Programming of Future Service-oriented Architectures. Journal of Software 4 (2009)

[8] Caeiro, M., Németh, Z., Priol, T.: A Chemical Model for Dynamic Workflow Coordination. In: PDP. pp. 215–222 (2011)

[9] Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.: Scribe: A Large-Scale and Decentralized Application-Level Multicast Infrastructure. IEEE JSAC 20 (2002)

[10] Huang, L., Tong, W., Kam, W., Sun, Y.: Implementation of GAMMA on a Massively Parallel Computer. JCST 12 (1997)

[11] Lin, H., Kemp, J., Gilbert, P.: Computing Gamma Calculus on Computer Cluster. IJTD 1(4), 42–52 (2010)

[12] Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In: Middleware. pp. 329–350 (2001)

[13] Schneider, F.B.: Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. ACM Comput. Surv. 22 (1990)

[14] Viroli, M., Zambonelli, F.: A Biochemical Approach to Adaptive Service Ecosystems. Information Sciences (2009)

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399